THE ADAPTIVE CRUISE CONTROLLER CASE STUDY

Visualisation, Validation, and Temporal Verification

Alexander Krupp[1], Ola Lundkvist[2], Tim Schattkowsky[1] and Colin Snook[3]
[1]Paderborn University; [2]Volvo Technology Corporation; [3]University of Southampton

Abstract: We present the adaptive cruise controller case study for B modelling and the model checking by RAVEN. Individual translations of B operations, data types, and invariants to the RAVEN Input Language are presented by the example of the case study.

Key words: aumotive systems, formal refinement, model checking, UML

1. INTRODUCTION

We have written a B specification for an adaptive cruise control following the lines of Chapter 5 and 6, respectively. The specification was verified as internally consistent but was difficult to capture even to those involved in its creation. Thus, an additional validation of the model was needed to convince the stakeholders that the model had the required behaviour. We therefore manually translated the model into a UML-B version in order to make it easier to understand and reason about. We automatically translated the UML-B model into a new B version using U2B and animated it using the ProB model checker/animator [5]. The animation did not reveal any problems with the model but substantially improved confidence and understanding in its behaviour.

In the process, some temporal requirements were deduced that had not been covered by the previous consistency verification. In order to check those properties, the B code produced by U2B was translated into RIL and verified by the RAVEN model checker [4]. For that, we manually translated the B code, which was generated by the UML2B tool to the RAVEN Input

Language (RIL) and verified the model and the specification through additional model checking. No problems were found during the temporal verification. As the structure of the existing B specification of the cruise controller is quite different from the approach, which was undertaken in Chapter 10 and in [2,3], we took a different approach in the translation to RIL. We have performed that complementary model checking of temporal properties since model checking additionally allows for reasoning about state sequences, which is expressible, but rather difficult to prove in B. The main advantages of the RAVEN model checker are:

- counter example generation for invalid sequences of operations,
- data analysis (min/max values of variables), and
- time analysis (min/max time of specific state configurations).

However, due to the state explosion problem, the main general drawback of model checking is of course the rather limited state space to be explored. Usually, abstraction techniques have to be applied to reduce the state space. As B offers a verification path from an abstract system towards an implementation, it seems natural to apply model checking at an abstract level in B, when a state space can be kept small.

The remainder of this chapter is structured as follows. The next two sections outline the translation of B operations, data types, and invariants to the RAVEN Input Language (RIL) and their visualisation as UML-B. Thereafter, verification results are presented.

2. VISUALISATION OF THE MODEL IN UML-B

To obtain an adequate visual UML-B presentation of the B model, we firstly examine B for indications of modelling of a set of instances. Typically, if several B operations have a parameter with the same type and this type is used in the domain of some function type variables, they can be combined to classes with class diagrams for representation. However, in the B model, no sets of instances are modelled in this way. Our second strategy was to look for a variable that is used as a guard and then changed to a new value by several operations. The life cycle of such a variable can be usefully represented as a state diagram with the operations at transitions. At first sight, there appeared to be no such variable, however, the invariant indicated that a group of Boolean variables that were used as guards in operations contained exclusions, e.g., `var1=F=>var2..n=F`. Such a group of variables was found to collectively represent a state variable. By replacing the group with a single variable representing the state, a better visual

representation from a different perspective was achieved. We discovered no disadvantages of combining these Booleans although we did not attempt a refinement. This representation generated a collation of related operations that all transitions the system between its main operational conditions such as `cruiseBecomesNotAllowed, cruiseOff` etc.

Similarly, a variable representing the state of the closest obstacle was identified resulting in a collation of operations related to the obstacle status. We decided on a natural segregation of the model into two classes as given in Figure 1.

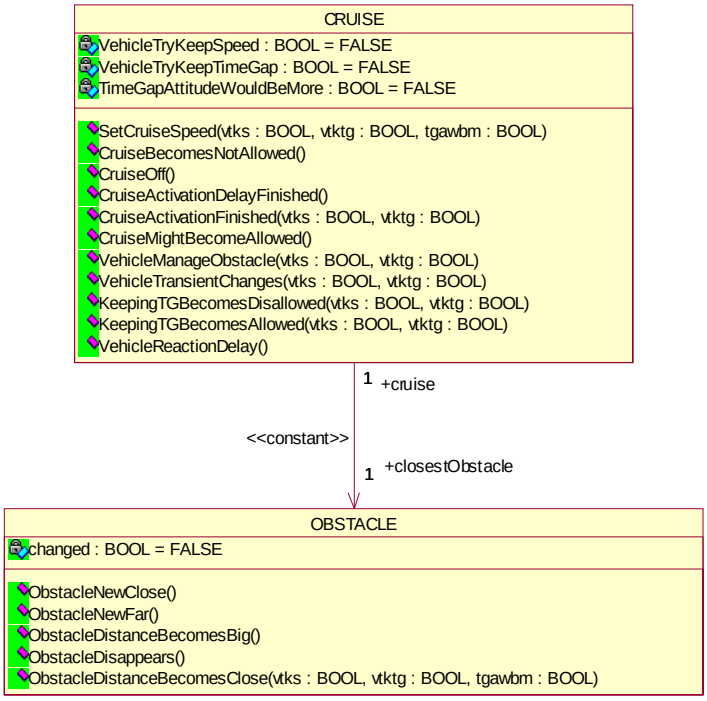

Figure 1: Class Diagram for the Cruise Controller

The classes are `CRUISE` and `OBSTACLE`. Note, that the case study, just has a single instance of each class. This enabled the use of an association to represent closestObstacle (since associations are links between instances), which either is an unnecessary complication compared to the original B model, or is necessary to create valid invariants for UML-B model. For each class, the behaviour is defined as a state diagram. Figure 2 gives the state diagram from class `OBSTACLE` and Figure 3 shows the corresponding B code with invariants.

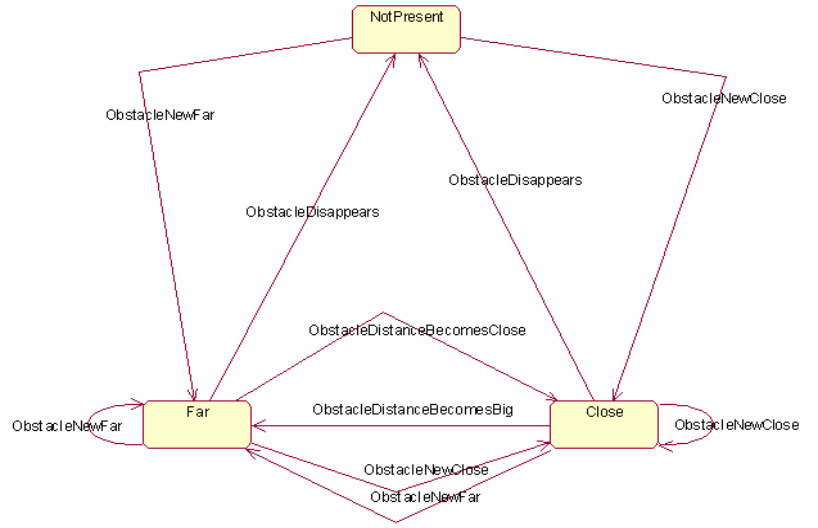

Figure 2: State Diagram for OBSTACLE

```
CARDINALITY 1
DEFINITIONS
CruiseReallyActive ==
       (closestObstacle.changed = FALSE &
        cruise_state/=CruiseActivationInProgress);
AccLaws(od, vtks, vtktg, tgawbm) ==
      /* do something law */
      (vtks = TRUE or vtktg = TRUE) &
      /* keep speed if far or no obstacle */
      (od :{ Far,NotPresent} => vtks = TRUE) &
      /* keep time gap otherwise, if not too fast */
      ((od = Close & tgawbm = FALSE) => vtktg = TRUE) &
      /* keep speed, if time gap
            keeping would be too fast */
      ((od = Close & tgawbm = TRUE) => vtks = TRUE)
INVARIANT
  (cruise_state:{CruiseNotAllowed, CruiseAllowed} =>
      VehicleTryKeepSpeed=FALSE) &
  (cruise_state:{CruiseNotAllowed, CruiseAllowed} =>
      VehicleTryKeepTimeGap=FALSE) &
  (cruise_state:{CruiseNotAllowed, CruiseAllowed} =>
      closestObstacle.changed=FALSE) &
  (cruise_state:{CruiseNotAllowed, CruiseAllowed} =>
      TimeGapAttitudeWouldBeMore=FALSE) &
  (closestObstacle.obstacle_state=NotPresent =>
      VehicleTryKeepTimeGap=FALSE) &
  (closestObstacle.obstacle_state:{NotPresent,Far} =>
      TimeGapAttitudeWouldBeMore=FALSE) &
  (cruise_state:{CruiseActive,CruiseActivationInProgress} &
      CruiseReallyActive =>
      (AccLaws(closestObstacle.obstacle_state,
      VehicleTryKeepSpeed,
      VehicleTryKeepTimeGap, TimeGapAttitudeWouldBeMore)
      )
)
INITIALISATION
closestObstacle:=thisOBSTACLE
```

Figure 3: UML-B Invariants

3. TRANSLATION OF B OPERATIONS AND DATA TYPES

In contrast to the automatic approach of Chapter 10 and 14, we take a pragmatic approach of a direct translation from the B model to the RAVEN Input Language (RIL) here. We outline how to translate the B model of Volvo's Adaptive Cruise Controller, generated from the UML2B tool to the RIL. For further information on that model and on the case study, please consult Chapter 10 or [3,4].

When translating from B to RIL directly, we have to start with the examination of the state space of the B model, i.e., we examine the set of B variables as given in the following enumeration.

B code

```
VARIABLES
  obstacle_state,
  changed,
  cruise_state,
  VehicleTryKeepSpeed,
  VehicleTryKeepTimeGap,
  TimeGapAttitudeWouldBeMore,
  NumberOfSetCruise
```

Additionally, the type of each variable from the B invariant has to be considered.

B code

```
obstacle_state: OBSTACLE_STATE &
changed: BOOL &
cruise_state: CRUISE_STATE &
VehicleTryKeepSpeed: BOOL &
VehicleTryKeepTimeGap: BOOL &
TimeGapAttitudeWouldBeMore: BOOL &
NumberOfSetCruise: NATURAL
```

The four Boolean variables in the B code translate to RIL easily. The enumerated set variables `OBSTACLE_STATE` and `CRUISE_STATE`, have rather small associated sets specified in the SETS clause as given hereafter.

B code

```
SETS
  OBSTACLE_STATE = {NotPresent, Close, Far};
  CRUISE_STATE   = {CruiseNotAllowed,
                    CruiseAllowed,
                    CruiseActive,
                    CruiseActivationInProgress};
```

Since the unrestricted natural variable `NumberOfSetCruise` would cause a state explosion with a BDD model checker, we additionally have to restrict the value of `NumberOfSetCruise` for model checking to a smaller interval, so that the translation the complete set of variables to RIL is as follows.

RIL code

```
MODULE Cruise
SIGNAL
  obstacle_state: NotPresent Close Far
  changed: BOOL
  cruise_state: {CruiseNotAllowed,
                 CruiseAllowed,
                 CruiseActive,
                 CruiseActivationInProgress}
  VehicleTryKeepSpeed: BOOL
  VehicleTryKeepTimeGap: BOOL
  TimeGapAttitudeWouldBeMore: BOOL
  NumberOfSetCruise: RANGE[0,4]
```

We can then continue with translation of the B operations starting with a set of B operations, which may be triggered any time their preconditions are valid. As each of the transitions executes due to a different set of source states, we cannot declare a specific set of variables as source states for the RAVEN transitions. Instead, we simply write *TRUE* as the source state for the transitions. The execution of each transition is then dependent on its *guard*. Generic transitions in RIL correspond to the following patterns:

```
|- [source state] -- [guard] --> [target state]
```

Consider the operation `ObstacleDisappears` as an example for translations of B operations to RIL.

B code

```
ObstacleDisappears =
  BEGIN
     SELECT obstacle_state=Close
     THEN obstacle_state:=NotPresent
     WHEN obstacle_state=Far
     THEN obstacle_state:=NotPresent
  END ||
  IF cruise_state : {CruiseActive,CruiseActivationInProgress}
  THEN
     changed := TRUE ||
     TimeGapAttitudeWouldBeMore := FALSE ||
     VehicleTryKeepTimeGap := FALSE
  END
```

The operation has two simultaneous substitutions. In RIL, we create one transition for each combination of choices in the simultaneous substitutions.

RIL code

```
// ObstacleDisappears
|- TRUE -- (obstacle_state=NotPresent) &
            !((cruise_state=CruiseActive) |
              (cruise_state=CruiseActivationInProgress))
        --> obstacle_state:=NotPresent
|- TRUE -- (obstacle_state=Close) &
            !((cruise_state=CruiseActive) |
              (cruise_state=CruiseActivationInProgress))
        --> obstacle_state:=NotPresent
|- TRUE -- (obstacle_state=Far) &
            !((cruise_state=CruiseActive) |
              (cruise_state=CruiseActivationInProgress))
        --> obstacle_state:=NotPresent
|- TRUE -- (obstacle_state=NotPresent) &
            ((cruise_state=CruiseActive) |
             (cruise_state=CruiseActivationInProgress))
        --> obstacle_state:=NotPresent; changed:=TRUE;
            VehicleTryKeepTimeGap:=FALSE;
            TimeGapAttitudeWouldBeMore:=FALSE
|- TRUE -- (obstacle_state=Close) &
            ((cruise_state=CruiseActive) |
             (cruise_state=CruiseActivationInProgress))
        --> obstacle_state:=NotPresent; changed:=TRUE;
            VehicleTryKeepTimeGap:=FALSE;
            TimeGapAttitudeWouldBeMore:=FALSE
|- TRUE -- (obstacle_state=Far) &
            ((cruise_state=CruiseActive) |
             (cruise_state=CruiseActivationInProgress))
        --> obstacle_state:=NotPresent; changed:=TRUE;
            VehicleTryKeepTimeGap:=FALSE;
            TimeGapAttitudeWouldBeMore:=FALSE
```

A scheme for translation of simultaneous choice substitutions can be formulated in generalized substitution notation as follows.

```
(g1 ⇒ [S1]) □ (g2 ⇒ [S2]) □ . . . □ (gn ⇒ [Sn])‖
(h1 ⇒ [T1]) □ (h2 ⇒ [T2]) □ . . . □ (hn ⇒ [Tn])
```

This is presented in RIL by creating one transition for each combination of simultaneous guards and their associated combination of substitutions as given in the following code. Note here, that substitutions need to be simple assignments, though.

RIL code

```
|- TRUE -- g1 & h1 --> S1; T1
|- TRUE -- g1 & h2 --> S1; T2
...
|- TRUE -- g1 & hn --> S1; Tn
|- TRUE -- g2 & h1 --> S2; T1
|- TRUE -- g2 & h2 --> S2; T2
...
|- TRUE -- g2 & hn --> S2; Tn
...
|- TRUE -- gn & h1 --> Sn; T1
|- TRUE -- gn & h2 --> Sn; T2
...
|- TRUE -- gn & hn --> Sn; Tn
```

As a more complex example, consider the following B operation. It takes two Boolean parameters. Additionally, it uses a B definition called `AccLaws`, which is expanded and adapted to RIL. The parameters `vtks`, and `vtktg` inflict a more substantial change on the generated RIL. As the operation precondition accepts any combination of parameter values at any time, we need to emulate the operation invocation by a separate module in RIL. This module contains representations of all variables used as parameters. With each execution step it sets the variables to an arbitrary configuration. That means, that each execution step in the RAVEN execution sequence offers all possible parameter input valuations. The B operation also uses a nested choice construct, which requires a slightly different translation than before.

B code

```
ObstacleDistanceBecomesBig (vtks,vtktg) =
PRE
  vtks:BOOL &
  vtktg:BOOL
THEN
  SELECT
     (CruiseReallyActive => AccLaws(Far, vtks, vtktg, FALSE))
  THEN
     SELECT obstacle_state=Close
     THEN obstacle_state:=Far
     END ||
     TimeGapAttitudeWouldBeMore := FALSE ||
     IF
        cruise_state: {CruiseActive, CruiseActivationInProgress}
      THEN
        VehicleTryKeepTimeGap := vtktg ||
        VehicleTryKeepSpeed := vtks
     END
  END
END
```

It can be observed in the B code, that there is a first `SELECT` statement enclosing the remainder of the operation. The guard of this `SELECT` is therefore in every generated RIL transition. The enclosed part is translated as described earlier. The additional guards generated from it are combined with the guard of the enclosing `SELECT` statement.[1]

RIL code

```
// ObstacleDistanceBecomesBig
 |- TRUE -- (CruiseReallyActive ->
             ((vtks|vtktg) & vtks & TRUE & TRUE)) &
             (obstacle_state=Close) &
             ((cruise_state=CruiseActive) |
             (cruise_state=CruiseActivationInProgress))
         --> obstacle_state:=Far;
             TimeGapAttitudeWouldBeMore:=FALSE;
             VehicleTryKeepTimeGap:=vtktg;
             VehicleTryKeepSpeed:=vtks
 |- TRUE -- (CruiseReallyActive ->
             ((vtks|vtktg) & vtks & TRUE & TRUE)) &
              (obstacle_state=Close) &
              !( (cruise_state=CruiseActive) |
              (cruise_state=CruiseActivationInProgress))
         --> obstacle_state:=Far;
             TimeGapAttitudeWouldBeMore:=FALSE
```

[1] Note that, corresponding to the B book [1], a `SELECT` substitution can become *infeasible* if it does not contain an `ELSE` part and it is possible that none of it guards hold. When a simultaneous substitution contains one infeasible substitution, then the whole substitution becomes infeasible. Our translation implicitly generates transitions for *feasible* cases only.

4. TRANSFORMATION OF B INVARIANTS

This section sketches how parts of B invariants can be transformed to RIL specifications for verification by model checking. A translatable invariant has to comply to the following rules.

1. It may only refer to identifiers which conform to RIL types: `INT`, `NAT`, `BOOL`, and enumerated `SETS`.
2. It may only use unary and binary operators supported by CCTL: ¬, ∧, ∨, ⇒,⇔, =, <, <=, >, >=

Each compliant invariant can be translated to RIL. The B operators have to be replaced by their RIL equivalents, of course, and some additional brackets may have to be introduced due to different operator precedence. The invariant is then prefixed in RIL with the AG temporal operator.

The following B code shows a fragment of the B invariant, which can be translated to RIL as two separate CCTL formulae.

B code

```
... &
(cruise state = CruiseNotAllowed =>
  (VehicleTryKeepSpeed = FALSE &
  VehicleTryKeepTimeGap = FALSE &
  changed = FALSE &
  TimeGapAttitudeWouldBeMore = FALSE &
  NumberOfSetCruise = 0)) &
(cruise state = CruiseAllowed =>
  (VehicleTryKeepSpeed = FALSE &
   VehicleTryKeepTimeGap = FALSE &
   changed = FALSE &
   TimeGapAttitudeWouldBeMore = FALSE &
   NumberOfSetCruise = 0))
& ...
```

The next code gives the translated invariants in combination with additional checks. The translated B invariants are contained in the two specifications *binv3* and *binv4*. In CTL, such expressions are called *safety conditions*. Specification *l1* is a reachability condition. It states, that it is always possible that the state with identifier `changed` can be eventually reached.

RIL code

```
SPEC
...
binv3 := AG
          ((Cruise.cruise_state =
            Cruise.CruiseNotAllowed) ->
              (!Cruise.VehicleTryKeepSpeed &
               !Cruise.VehicleTryKeepTimeGap &
               !Cruise.changed &
               !Cruise.TimeGapAttitudeWouldBeMore &
               (Cruise.NumberOfSetCruise = 0)))

binv4 := AG
         ((Cruise.cruise_state =
           Cruise.CruiseAllowed) ->
             (!Cruise.VehicleTryKeepSpeed &
              !Cruise.VehicleTryKeepTimeGap &
              !Cruise.changed &
              !Cruise.TimeGapAttitudeWouldBeMore &
              (Cruise.NumberOfSetCruise = 0))

l1 := AG EF Cruise.changed
...
```

5. RESULTS

The verification with RAVEN has shown, that all invariants translated from B are correct. The overall execution time of the translated RIL example for RAVEN model checking and analysis was less than a second on an Intel P4, 2.2GHz, 1GB RAM. Memory allocation of RAVEN was approximately 3.9MB.

In addition to checking restricted B invariants, RAVEN is able to check constraints, which involve state sequences. Among those are additional qualitative properties like safety, reachability, response, inevitability, and liveness constraints for state-based verification. Especially with Event-B it is useful to be able to check for possible sequences of states. In addition, RAVEN performs quantitative and timing analysis. It allows to determine minimum or maximum reaction times, and to check for minimum and maximum values of integers within a certain set of states. Some examples of quantative data analysis can be found in Chapter 14 of this book.

However, our focus here was on a specification flow from B generated by UML2B to RAVEN in contrast to Chapter 10 and 14, where we investigate B patterns to verify the behavior of finite state machines. The feasibility of this approach was shown under specific restrictions on B.

Nevertheless, RAVEN's analysis capabilities give a good example that it makes sense to use B in combination with other formal verification means like model checking, which gives complementary verification and additional confidence in the model under design.

REFERENCES

1. J.R. Abrial. The B-Book. Cambridge University Press, 1996.
2. A. Krupp, W. Mueller. Deliverable D4.3.1: Specification of Real-Time Properties. Technical Report, IST - Project PUSSEE, December 2002.
3. A. Krupp, W. Mueller. Deliverable D4.3.2: Refinement and Verification of Real-Time Properties. Technical Report, IST - Project PUSSEE, April 2003
4. J. Ruf. RAVEN: Real-Time Analyzing and Verification Environment. J.UCS, Springer, Heidelberg, 2001.
5. P.J. Turner, M.A. Leuschel. ProB V1.06 User Manual. Declarative Systems and Software Engineering, University of Southampton, UK, 2004.
6. J. Zandin. Non-Operational, Temporal Specification Using the B Method – A Cruise Controller Case Study. Master's Thesis, Chalmers University of Technology and Gothenburg University, 1999.